**Stress-Map Documentation**

Pre-process extract

Pre-process the extract and start a routing engine HTTP server on port 5000.

Depending on what kind of routing you want to use, choose the profile accordingly:

car: car.lua

walking: foot.lua

bike: bicycle.lua

***docker run -t -v c:/docker:/data osrm/osrm-backend osrm-extract -p /opt/PROFILE.lua /data/CITY-latest.osm.pbf***

So for car rides in Berlin use:***docker run -t -v c:/docker:/data osrm/osrm-backend osrm-extract -p /opt/car.lua /data/berlin-latest.osm.pbf***

Then run:***docker run -t -v c:/docker:/data osrm/osrm-backend osrm-partition /data/berlin-latest.osrm***

***docker run -t -v c:/docker:/data osrm/osrm-backend osrm-customize /data/berlin-latest.osrm***

**Why we setup Docker Image**

Because extracting, customizing osm.pbf that is map of any particular region is not suitable for desktop machine requires lot of computational power and RAM so Docker

**How to Setup OSRM Docker Image**

1. Refer this article

<https://gist.github.com/AlexandraKapp/e0eee2beacc93e765113aff43ec77789>

* 1. Start by installing docker on your desktop
  2. Pulling the image
  3. Download OpenStreetMap extract (:I downaloaeded onlot Atlanta, I suggest to downalod all of United States of America - <https://download.geofabrik.de/north-america/us.html>
  4. Start Pre-Processing since it is important for to create paths for Bicycles, Foot and Car
  5. Start Routing Engine  
       
     docker run --name osrm -t -i -p 5000:5000 -v c:/docker:/data osrm/osrm-backend osrm-routed --algorithm mld /data/berlin-latest.osrm

* 1. To test request a based on profile use processed change driving to desired

curl "<http://127.0.0.1:5000/route/v1/driving/13.388860,52.517037;13.385983,52.496891?steps=true>"

* 1. Refer Installation of OSRM Server document pdf if faced with any issues
  2. Docker start “nameofyourdocker” Docker stop “nameofyourdocker”

Additional References <https://medium.com/ula-engineering/getting-started-with-osrm-a-guide-1854891fff11>

1. Other alternatives Valhalla / Mapbox / Graphhopper
   1. Documentation

**OSRM Open Source Routing Machine API**

OSRM Response

{

"code": "Ok", **// Indicates the status of the response (successful)**

"matchings": [ **// An array of matched routes**

{

"confidence": 0.922436, /**/ Confidence level of the match**

"geometry": { /**/ Route geometry**

"coordinates": [ **// Array of coordinates defining the route**

[-84.365211, 33.777663], **// Longitude and latitude of the first coordinate**

[-84.364595, 33.777672] **// Longitude and latitude of the second coordinate**

],

"type": "LineString" **// Type of geometry (LineString)**

},

"legs": [ **// Information about route legs**

{

"annotation": { /**/ Annotation for this leg**

"metadata": { **// Metadata about the annotation**

"datasource\_names": [ **// Names of data sources used**

"lua profile"

]

},

"nodes": [69244349, 1940792548], **// Nodes along the route**

"datasources": [0], **// Data sources used**

"speed": [4.2]**, // Speed of the route (in meters per second)**

"weight": [13.6], /**/ Weight of the route**

"duration": [13.6], **// Duration of the route (in seconds)**

"distance": [56.958883] **// Distance of the route (in meters)**

},

"steps": [], **// Steps along this leg (empty in this case)**

"distance": 57, **// Total distance of the leg (in meters)**

"duration": 13.6, **// Total duration of the leg (in seconds)**

"summary": "", **// Summary of the leg (empty in this case)**

"weight": 13.6 **// Weight of the leg**

}

],

"distance": 57, /**/ Total distance of the matched route (in meters)**

"duration": 13.6, **// Total duration of the matched route (in seconds)**

"weight\_name": "duration", /**/ Name of the weight (duration in this case)**

"weight": 13.6 **// Weight of the matched route**

}

],

"tracepoints": [ **// Tracepoints representing the matched coordinates**

{

"alternatives\_count": 85, **// Number of alternative matches**

"waypoint\_index": 0, /**/ Index of the waypoint**

"matchings\_index": 0, /**/ Index of the matching**

"location": [-84.365211, 33.777663], /**/ Longitude and latitude of the tracepoint**

"name": "Drewry Street Northeast", **// Name of the street at the tracepoint**

"distance": 13.754717, **// Distance from the original coordinates to this tracepoint (in meters)**

"hint": "218zgO5fMwBIAAAAkAAAAAAAAAAnAAAAaVjyQSQhcEIAAAAA\_UuDQUgAAACQAAAAAAAAACcAAAAUAAAAZbD4-v9nAwJjsPj6e2gDAgAAnwXlPuR3" **// Hint for snapping to a street segment**

},

{

"alternatives\_count": 207,

"waypoint\_index": 1,

"matchings\_index": 0,

"location": [-84.364595, 33.777672],

"name": "Drewry Street Northeast",

"distance": 0.776406,

"hint": "218zgO5fMwDQAAAACAAAAAAAAAAnAAAANbmuQqi0PUAAAAAA\_UuDQdAAAAAIAAAAAAAAACcAAAAUAAAAzbL4-ghoAwLNsvj6D2gDAgAAnwXlPuR3"

}

]

}

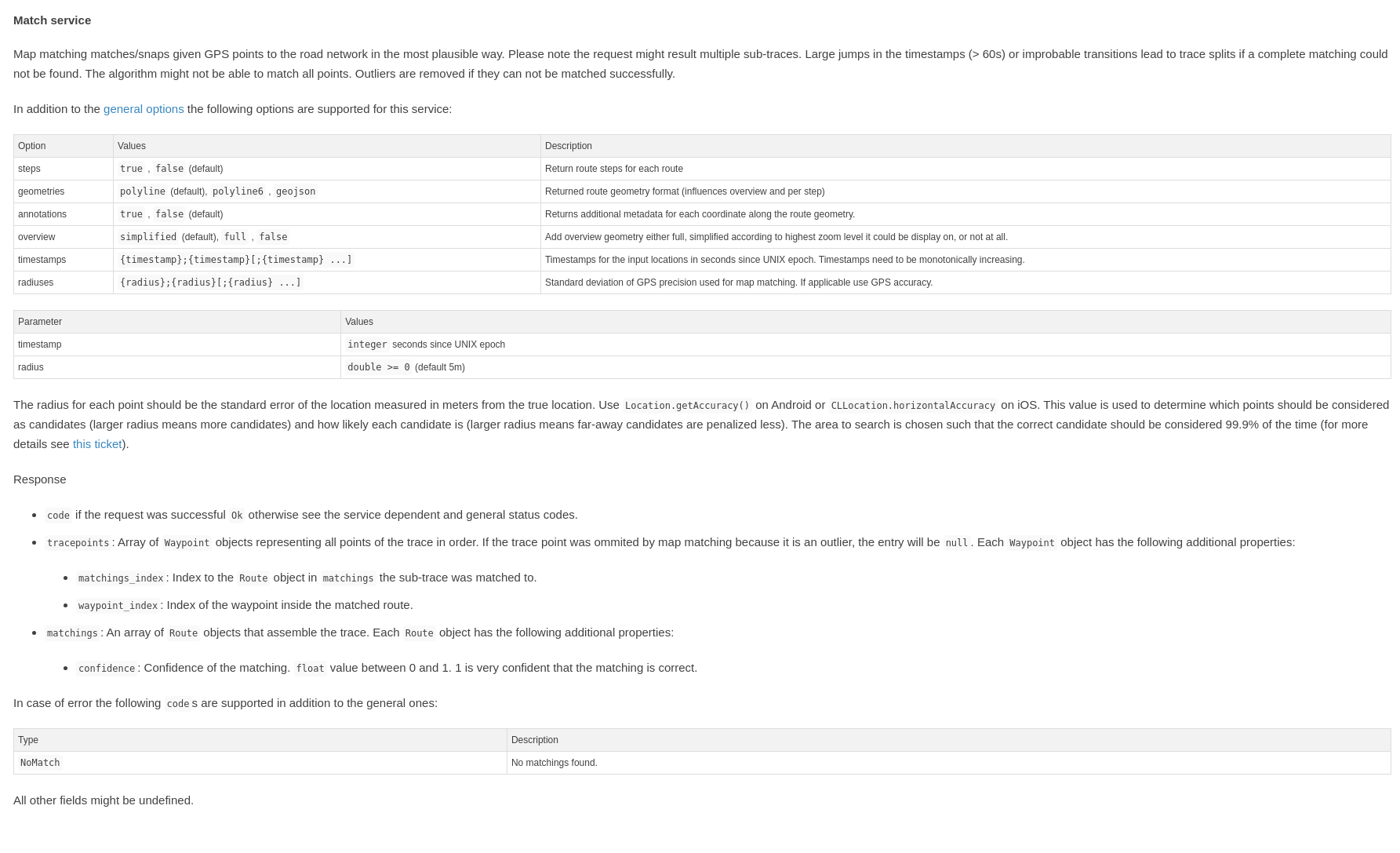
**For strucuture of the responses - [https://project-osrm.org/docs/v5.5.1/api/#result-objects](https://project-osrm.org/docs/v5.5.1/api/" \l "result-objects)**

1. OSRM documentation can be found on this link - [https://project-osrm.org/docs/v5.5.1/api/#general-options](https://project-osrm.org/docs/v5.5.1/api/" \l "general-options)

[https://www.jawg.io/docs/apidocs/routing/osrm](https://www.jawg.io/docs/apidocs/routing/osrm/)

* 1. **Match Service** - Map matching matches/snaps given GPS points to the road network in the most plausible way. Please note the request might result multiple sub-traces. Large jumps in the timestamps (> 60s) or improbable transitions lead to trace splits if a complete matching could not be found. The algorithm might not be able to match all points. Outliers are removed if they can not be matched successfully.

[https://project-osrm.org/docs/v5.5.1/api/#match-service](https://project-osrm.org/docs/v5.5.1/api/" \l "match-service)

We used match service to align the group of coordiantes from SQL server to route it (snapped) most efficient way using this service. It takes maximum of 90 to 100 coordiantes in one API call.

Example

<http://localhost:5000/match/v1/bicycle/-84.365213,33.777787;-84.364595,33.777679?overview=full&radiuses=49;49&geometries=geojson&tidy=true&annotations=true>



Annotations: it consists of metadata about the coordinates revived from API that is data about the snapped coordinates to the road. It inlcudes nodes,

"nodes": [2256317495, 69331811, 8155111912],

"datasources": [0, 0],

"speed": [4.2, 4.2],

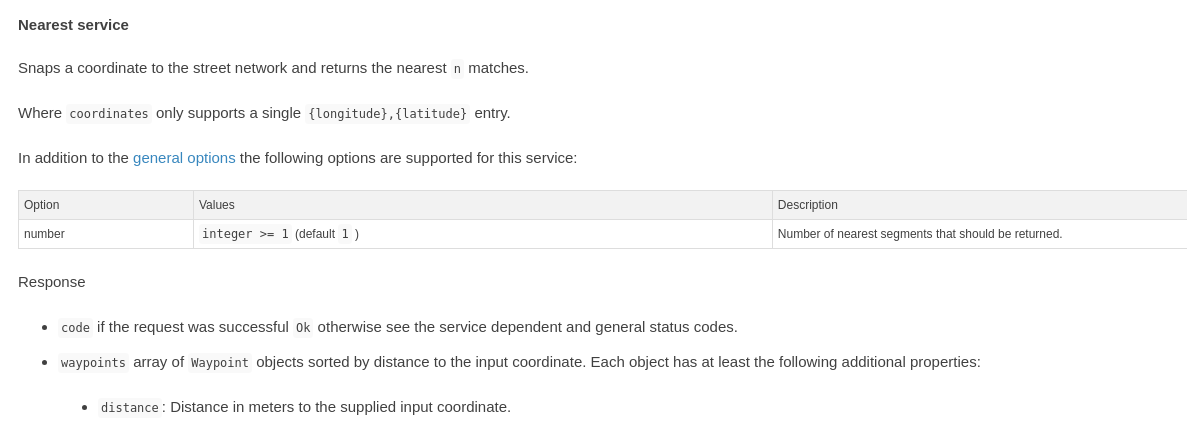
"weight": [14.5, 7.4],

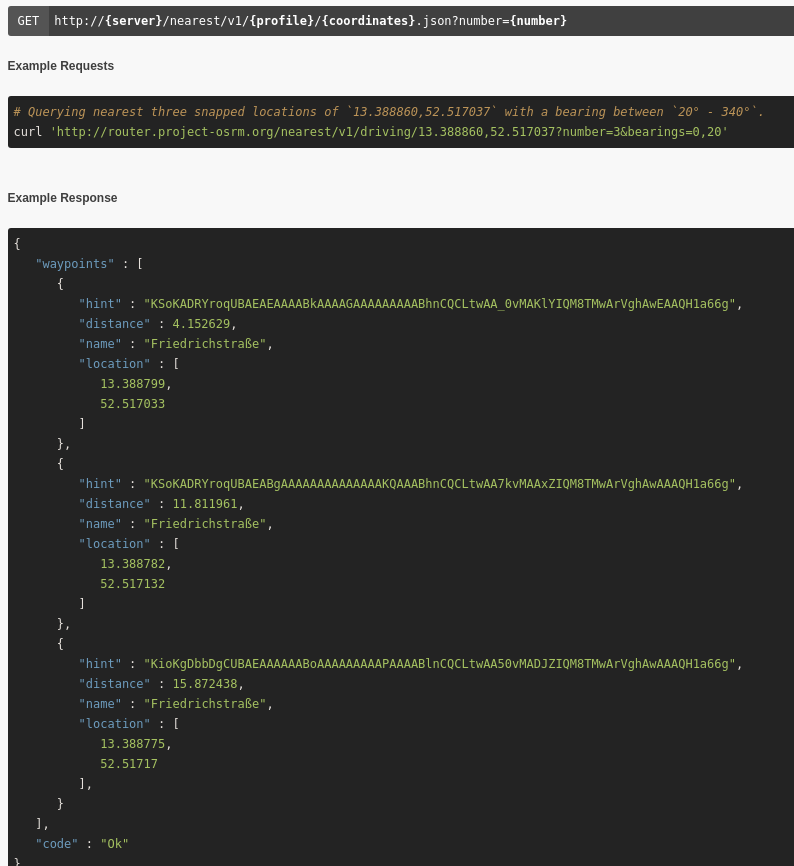
"duration": [14.5, 7.4],

"distance": [60.380708, 31.030934]

Properties

* distance: The distance, in metres, between each pair of coordinates
* duration: The duration between each pair of coordinates, in seconds
* datasources: The index of the datasource for the speed between each pair of coordinates. 0 is the default profile, other values are supplied via --segment-speed-file to osrm-contract
* nodes: The OSM node ID for each coordinate along the route, excluding the first/last user-supplied coordinates
  1. Nearest Service – It could be used to find out nearest ways, roads, paths here bearing is the angle in which you want to search the nearest street network coordinate



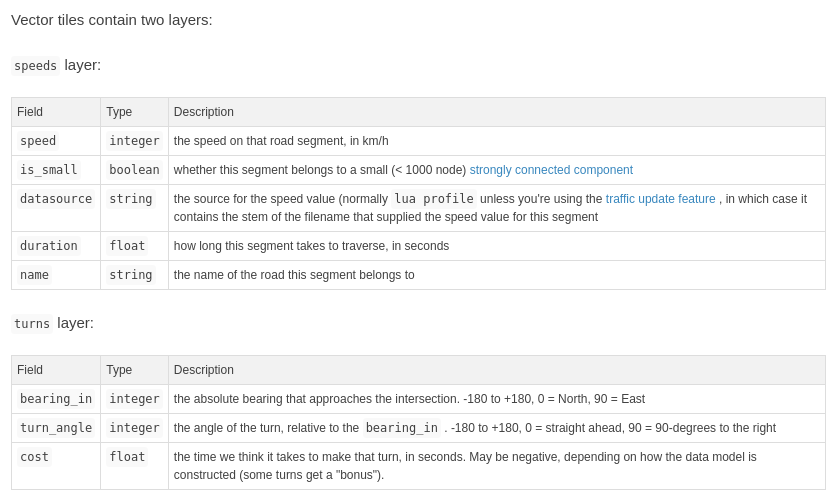


* 1. Route Service - Provides functionality to calculate routes between two or more locations, considering various factors like distance, duration, traffic conditions, etc.
     1. Applications: Navigation and routing apps: Helps users find the shortest or fastest route between two points, considering traffic conditions and other constraints. Logistics and fleet management: Optimizes delivery routes to minimize time and fuel consumption, improving efficiency. Emergency services: Helps emergency responders find the quickest route to reach a location in case of emergencies.
  2. Trip Service The trip plugin solves the Traveling Salesman Problem using a greedy heuristic (farthest-insertion algorithm). The returned path does not have to be the fastest path, as TSP is NP-hard it is only an approximation. Note that if the input coordinates can not be joined by a single trip (e.g. the coordinates are on several disconnected islands) multiple trips for each connected component are returned.

[https://project-osrm.org/docs/v5.5.1/api/#trip-service](https://project-osrm.org/docs/v5.5.1/api/" \l "trip-service)

Manages and analyzes trip data, including GPS traces, timestamps, and other relevant information.

* + 1. Applications: Transportation analysis: Analyzes travel patterns and behaviors to understand traffic flow, congestion hotspots, and commuting patterns. Tourism planning: Helps tourism agencies analyze visitor movement patterns to optimize tourist routes and attractions placement. Environmental impact assessment: Studies the impact of transportation on the environment by analyzing trip data to identify areas with high emissions or congestion.
  1. Tile Service Provides raster map tiles for displaying maps on web or mobile applications.

This service generates Mapbox Vector Tiles that can be viewed with a vector-tile capable slippy-map viewer. The tiles contain road geometries and metadata that can be used to examine the routing graph. The tiles are generated directly from the data in-memory, so are in sync with actual routing results, and let you examine which roads are actually routable, and what weights they have applied. The x, y, and zoom values are the same as described at https://wiki.openstreetmap.org/wiki/Slippy\_map\_tilenames, and are supported by vector tile viewers like Mapbox GL JS. The response object is either a binary encoded blob with a Content-Type of application/x-protobuf, or a 404 error. Note that OSRM is hard-coded to only return tiles from zoom level 12 and higher (to avoid accidentally returning extremely large vector tiles).

[https://project-osrm.org/docs/v5.5.1/api/#tile-service](https://project-osrm.org/docs/v5.5.1/api/" \l "tile-service)

* + 1. Applications: Web mapping applications: Displays interactive maps on websites and mobile apps, allowing users to explore geographic data and visualize information. Location-based services: Integrates maps into applications to provide location-aware features like finding nearby restaurants, businesses, or points of interest. Geographic information systems (GIS): Supports spatial analysis and visualization by providing map layers as raster tiles, enabling users to overlay and analyze different datasets.
  1. Table Service - Computes the duration of the fastest route between all pairs of supplied coordinates. Manages and stores tabular data associated with geographic features, such as attributes of map features or spatial data.

[https://project-osrm.org/docs/v5.5.1/api/#table-service](https://project-osrm.org/docs/v5.5.1/api/" \l "table-service)

* + 1. Applications: Data management: Stores and organizes geographic data in structured tables, allowing efficient retrieval and management of spatial information. Geographic data analysis: Provides a platform for analyzing and querying spatial data attributes, enabling users to derive insights and make data-driven decisions. Asset management: Stores information about assets like infrastructure, utilities, or property boundaries, facilitating asset tracking, maintenance, and planning. Land management: Supports land administration by storing information about land parcels, ownership, and land use, helping governments and organizations manage land resources effectively.

**OSM Open Street Map API**

1. Open Street -

Communities to Join - https://www.openstreetmap.org/communities

https://openstreetmap.us/

Reference API - <https://publicapis.io/open-street-map-api>

To verify or find a way check my insert way id <https://www.openstreetmap.org/way/9186521>

9186521 is way id here.

For a node to check the marker - https://www.openstreetmap.org/node/6016047119

6016047119 enter desired node id instead

For a relation <https://www.openstreetmap.org/relation/13432933>

A way is one of the fundamental elements of the map. In everyday language, it is a line. A way normally represents a linear feature on the ground (such as a road, wall, or river).

A tag consists of two items, a key and a value. Tags describe specific features of map elements (nodes, ways, or relations) or changesets.

For OSM Docker - https://switch2osm.org/serving-tiles/using-a-docker-container/

Overpass API that uses OSM - https://wiki.openstreetmap.org/wiki/Overpass\_API

Demo - <https://overpass-turbo.eu/>

Wiki - [https://wiki.openstreetmap.org/wiki/Overpass\_API/Overpass\_QL#out](https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL" \l "out)

Overpass Documentation - https://readthedocs.org/projects/python-overpy/downloads/pdf/latest/

Relation example - [https://overpass-turbo.eu/?Q=relation(2081626)%3B%3E%3Bout%3B&C=48.12601;11.5668;13&R](https://overpass-turbo.eu/?Q=relation(2081626)%3B>%3Bout%3B&C=48.12601;11.5668;13&R)

For Nodes

node\_query = op.query("""

[out:json];

node({node\_id});

out center;

""".format(node\_id=node\_id))

For Ways

ways\_query = op.query("""

[out:json];

node({node\_id});

<;

out;

""".format(node\_id=node\_id))

**Architecture**

1. Polyglot
   1. Python & Py Notebooks
   2. Express, Node Javascript / Typescript
   3. SQL

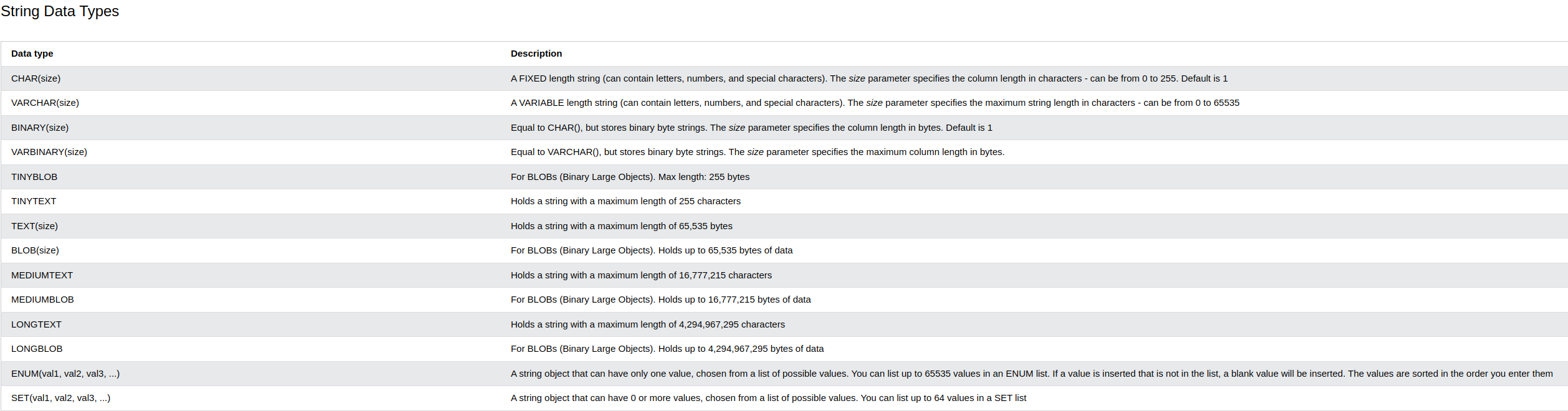
|  |  |  |  |
| --- | --- | --- | --- |
| Data Extraction and Analysis | Python & Py Notebooks (OSRM-API.pynb / OSM-API.pynb) | OSRM API for routing | OSM API for Nodes |
| MySQL Table (st – database, osr table, osm table) | | |
| Translation // Backend | Node & Express Javascript Loading MySQL table (server.js), | | |
| Frontend | Front end display of routes (MyMap.tsx, Snap.tsx) | | |

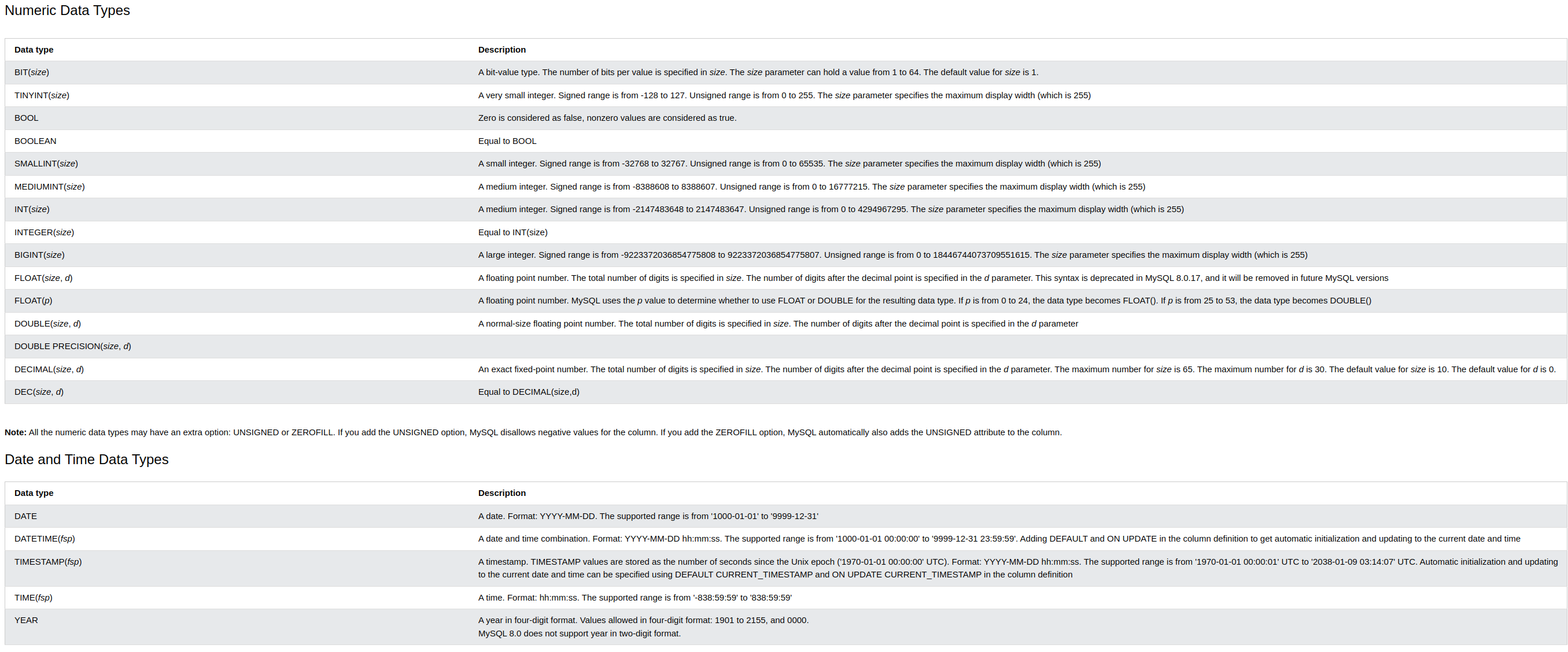
CI/CD Pipeline

**SQL Table**

I am writing about datatypes here that we used to store the information from OSRM & OSM TEXT, LONGTEXT, VARCHAR, INT.

Below screenshots are about the description and size of data types for future reference.





**To start MySQL on Linux:**

**My System:** sudo mysql -u root -p (password is root)

**if not root just** press enter

USE DatabaseName;

Here, the **DatabaseName** is the name of the database that we want to select

CREATE DATABASE testDB;

SHOW DATABASES;

**to input database:** “soruce /filepath”

**To install & start XAMPP:**

How do I install XAMPP?

Choose your flavor for your linux OS, the 32-bit or 64-bit version.

Change the permissions to the installer

chmod 755 xampp-linux-\*-installer.run

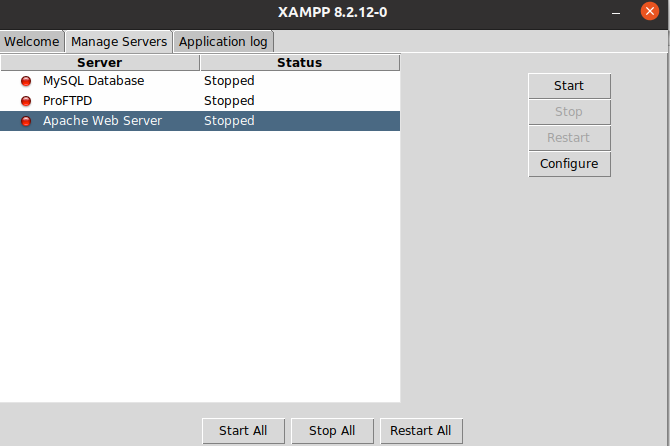
Run the installer

sudo ./xampp-linux-\*-installer.run

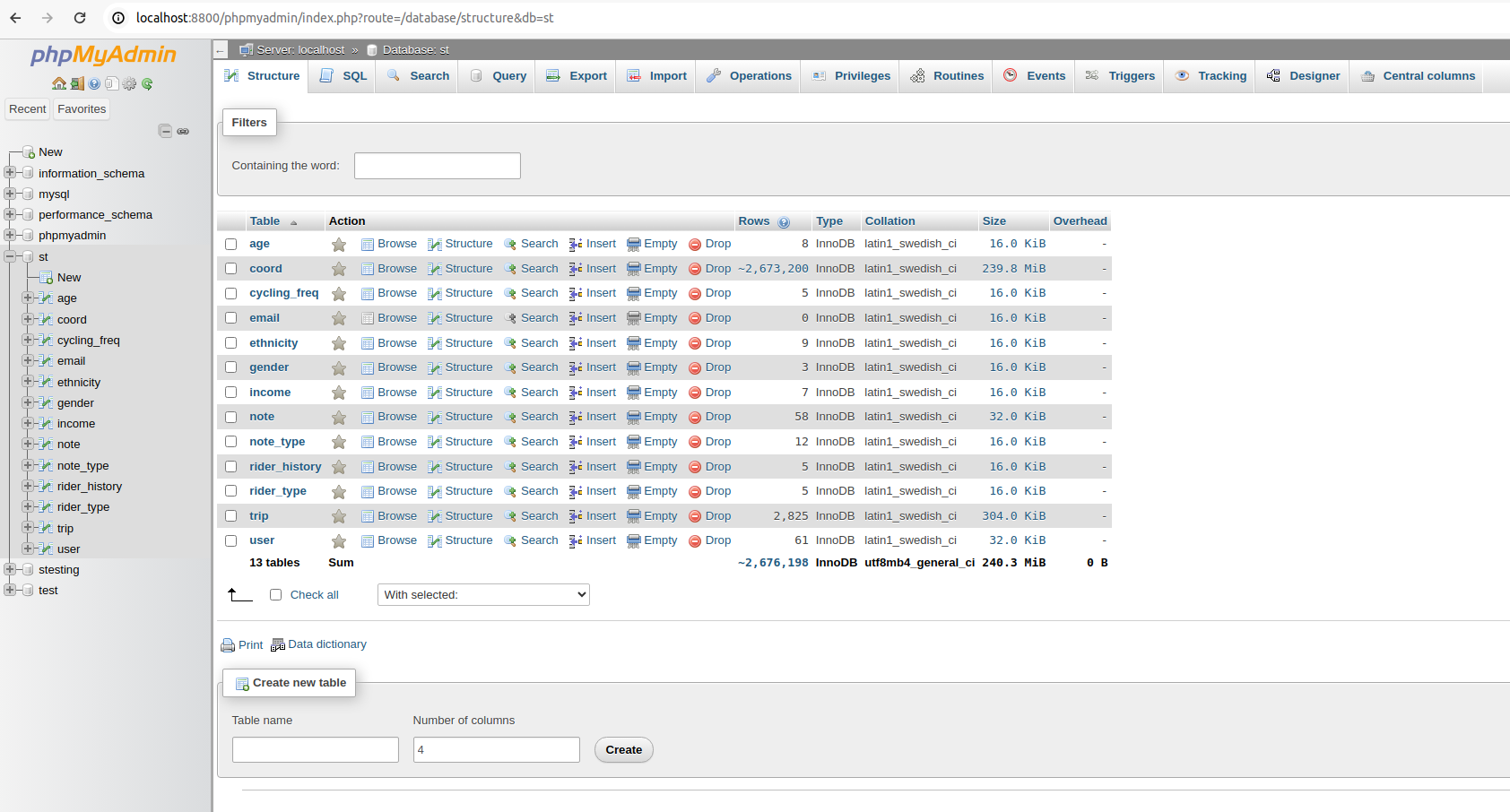
That's all. XAMPP is now installed below the /opt/lampp directory.

cd *opt*lampp/

sudo ./manager-linux-x64.run



start Apache Web Server and start MySQL Database then go to localhsot phpyadmin to access and see the SQL table



**To access XAMPP/MariaDB through terminal on Linux:**

“sudo /opt/lampp/bin/mysql -h localhost -u root -p”

**to input database:** “soruce /filepath”

**CODE**

**Snap.tsx**

The Snap component is a React functional component used to display trip data on an interactive map. It fetches trip data from a server, retrieves geographic coordinates, and plots them on a Leaflet map. This documentation provides a detailed explanation of the component's structure, functionality, and logic.

#### Dependencies:

* React: A JavaScript library for building user interfaces.
* Axios: A Promise-based HTTP client for making requests to the server.
* Leaflet: An open-source JavaScript library for interactive maps.
* CSS: Styling for the map and loading animation.

#### Component Structure:

1. State Variables:
   * trip: Stores the fetched trip data.
   * step: Represents the window size for dividing trip data into parts.
   * tripId: Represents the selected trip ID.
   * mapRef: Ref for the Leaflet map instance.
   * mapError: Holds error messages.
   * loading: Tracks loading status.
2. Effects:
   * Fetches trip data from the server based on the selected trip ID.
   * Divides trip data into parts and processes each part to retrieve coordinates and plot them on the map.
3. Event Handlers:
   * handleStepChange: Updates the window size (step) based on the range input.
   * handleTripIdChange: Updates the selected trip ID based on the dropdown selection.
4. Rendering:
   * Renders the Leaflet map and UI elements.
   * Displays a range input for adjusting the window size.
   * Provides a dropdown menu for selecting trip IDs.
   * Shows loading animation during data fetching.
   * Displays error popups for error messages.

#### Functionality:

1. Fetching Trip Data:
   * Uses Axios to fetch trip data from the server based on the selected trip ID.
   * Updates the trip state variable with the fetched data.
2. Plotting Coordinates:
   * Divides trip data into parts based on the window size (step).
   * Processes each part to retrieve coordinates using the OSRM API.
   * Plots the coordinates on the Leaflet map as polylines and markers.
3. User Interaction:
   * Allows users to adjust the window size and select trip IDs.
   * Updates the map and displayed data accordingly based on user interactions.
4. Error Handling:
   * Handles errors during data fetching and API requests.
   * Displays error messages to the user.

#### Conclusion:

The Snap component provides a user-friendly interface for visualizing trip data on an interactive map. It offers flexibility in adjusting the window size and selecting specific trip IDs for analysis. With error handling and loading animations, it ensures a smooth user experience even during data retrieval.

**MyMap.tsx**

The MyMap component is a React functional component designed to display trip data on a Leaflet map. It fetches trip and user information from a backend server, allowing users to select specific trips and view associated details. Additionally, it provides filters for refining displayed data based on various attributes.

### Dependencies

* React: The component is built using React, a JavaScript library for building user interfaces.
* axios: This library is used for making HTTP requests to fetch trip and user data from the server.
* Leaflet: Leaflet is an open-source JavaScript library for interactive maps.
* typescript: TypeScript is used to enable static type-checking and improved developer tooling.

### Interfaces

The component defines several TypeScript interfaces to maintain type safety:

* Coordinates: Represents geographic coordinates with a latitude, longitude, and trip\_id.
* Trip: Describes the structure of trip data fetched from the server, including code and matchings.
* TripInfo: Contains information about each trip, such as id, n\_coord, purpose, start, stop, and user\_id.
* UserInfo: Represents user information with properties like user\_id, agetype, cycling\_freqtype, etc.

### State Variables

* trips: Stores trip data fetched from the server.
* currentTripIds: Tracks the start and end trip IDs displayed on the map.
* selectedTripId: Stores the ID of the currently selected trip.
* tripinfo: Holds information about all trips.
* selectedTripInfo: Contains details of the selected trip.
* currentPage: Tracks the current page number for pagination.
* userinfo: Stores user information fetched from the server.
* selectedUserInfo: Holds details of the selected user.
* attributeTypes: Stores unique values for various user attributes.
* showFilters: Indicates whether the filters section is visible.

### useEffect Hooks

* Fetch Trips: Fetches trip data from the server based on the current page and updates the trips state.
* Fetch Trip Info: Retrieves information about trips from the server and updates the tripinfo state.
* Fetch User Info: Fetches user information from the server and updates the userinfo state. It also extracts unique attribute values and updates the attributeTypes state.
* Map Initialization: Initializes the Leaflet map and plots trip routes based on trips, currentTripIds, and selectedTripId.

### Helper Functions

* plotPolyline: Plots trip routes on the map using Leaflet. It takes trip data, start and end trip IDs, and the selected trip ID as input.
* processOsrmData: Processes OSRM API response data to extract coordinates for plotting on the map.
* getRandomColor: Generates a random color for styling trip routes on the map.

### Event Handlers

* handleNextPage: Increments the current page number for pagination.
* handlePreviousPage: Decrements the current page number for pagination.
* handleTripSelection: Updates the selected trip ID and fetches associated trip information from tripinfo.
* handleUserSelection: Updates the selected user ID and fetches associated user information from userinfo.

### JSX Elements

* Map Display: Renders the Leaflet map component with trip routes plotted.
* Trip and User Selection: Provides dropdowns for selecting trips and users, allowing users to view detailed information.
* Filters: Displays filters for refining displayed data based on various attributes. The filter section can be toggled on/off.

### Conclusion

The MyMap component is a comprehensive solution for visualizing trip data on an interactive map. It fetches data from the server, allows users to select specific trips and users, and provides filtering options for refining displayed information. With its clean and organized structure, developers can easily understand and extend its functionality.

In TypeScript, the interface keyword is used to define custom data types, which specify the shape of objects. It allows developers to define the structure of objects and enforce type checking at compile time.

In the provided example, the interface Trip defines the structure of objects that represent trip data. It specifies that a Trip object must have the following properties:

* code: A string property.
* matchings: An array property containing objects with a specific structure, which includes a geometry property containing an array of arrays of numbers.

By defining this interface, TypeScript ensures that any object labeled as a Trip must adhere to this structure. This helps catch potential errors during development, as TypeScript checks whether objects conform to their defined interfaces and provides helpful error messages if they do not.

**OSRM-API.pynb (cycle-stress-map/playground/OSRM-API.ipynb)**

#### Purpose:

This Python script is designed to interact with the OSRM (Open Source Routing Machine) API to match GPS coordinates from trip data and retrieve route information.

#### Functions:

1. create\_pairs(data):
   * Purpose: This function generates pairs of latitude and longitude coordinates from the provided trip data.
   * Parameters:
     + data: A DataFrame containing trip data with latitude and longitude columns.
   * Returns:
     + A list of pairs of latitude and longitude coordinates.
2. cpmini(data):
   * Purpose: Similar to create\_pairs, this function generates pairs of latitude and longitude coordinates. However, it starts from the first data point and includes all data points in the trip.
   * Parameters:
     + data: A DataFrame containing trip data with latitude and longitude columns.
   * Returns:
     + A list of pairs of latitude and longitude coordinates.
3. send\_osrm\_request(pairs, trip\_id, speed, recorded):
   * Purpose: Sends a request to the OSRM API to match the provided pairs of coordinates and retrieves route information.
   * Parameters:
     + pairs: A list of pairs of latitude and longitude coordinates.
     + trip\_id: The unique identifier for the trip.
     + speed: The speed recorded for each coordinate in the trip.
     + recorded: The recorded time for each coordinate in the trip.
   * Returns:
     + A list of dictionaries containing matched coordinate information, including trip ID, response from the API, speed, recorded time, and node IDs.
   * Note: This function also handles errors by printing error messages and adding the trip ID to an error list.

Function Definitions:

* create\_pairs(data):
  + This function takes a DataFrame data as input.
  + It creates pairs of latitude and longitude coordinates from the DataFrame.
  + Pairs are created by selecting every 5th latitude and longitude starting from index 4.
  + The function returns a list of coordinate pairs.
* cpmini(data):
  + Similar to create\_pairs(data) but includes all latitude and longitude coordinates without skipping.
* send\_osrm\_request(pairs, trip\_id, speed, recorded):
  + Sends a request to the OSRM API with the provided coordinate pairs.
  + Constructs the API request URL based on the pairs, trip ID, speed, and recorded time.
  + Handles the API response, extracting relevant information such as matched coordinates and node IDs.
  + Appends the response data to the matched\_coordinates list.
  + If an error occurs, it logs the error and adds the trip ID to the error\_trip\_ids list.

3. Main Execution:

* Batch Processing:
  + The script processes trip data in batches based on the number of rows.
  + If the batch size is less than or equal to 9, the data is processed without splitting.
  + For larger datasets, the data is split into multiple DataFrames using numpy.array\_split.
* Sending Requests:
  + For each batch of data, the script constructs pairs of coordinates using create\_pairs.
  + If the batch size is small, it uses cpmini to include all coordinates without skipping.
  + The script then sends requests to the OSRM API for each batch.
* Error Handling:
  + If the API response status code is not 200, it logs the error and adds the trip ID to error\_trip\_ids.

1. main:
   * Purpose: The main execution function that processes trip data, generates coordinate pairs, and sends requests to the OSRM API.
   * Steps:
     + Iterate through each unique trip ID in the dataset.
     + Check if the trip data contains only one data point. If so, directly generate the response with the single coordinate.
     + If the trip data contains multiple data points:
       - Split the data into smaller batches based on the number of data points.
       - For each batch:
         * If the batch size is small, generate coordinate pairs and send a request to the OSRM API.
         * If the batch size is large, further split the data into smaller chunks to avoid exceeding API request limits.
     + Print the response from the OSRM API, including matched coordinates and any error messages.

#### Global Variables:

* matched\_coordinates: A list to store matched coordinate information retrieved from the OSRM API.
* error\_trip\_ids: A list to store trip IDs for which errors occurred during API requests.

#### Usage:

* To use this script, provide trip data in the form of a DataFrame containing latitude, longitude, trip ID, recorded time, and speed columns.
* Ensure that the OSRM API is accessible and running at the specified URL (http://localhost:5000/match/v1/bicycle).
* Execute the script, which will process the trip data, send requests to the OSRM API, and print the response.

**OSM-API.pynb (cycle-stress-map/playground/OSM-API.ipynb)**

This documentation provides an overview of the Python code used to retrieve geographic information using the Overpass API. The code extracts data for each node ID associated with trip records and constructs a DataFrame containing relevant information.

#### Dependencies

* Python: The code is written in Python.
* overpy: This library provides a Python interface for querying the Overpass API.
* pandas: It is a powerful data manipulation library used to handle data in tabular form.
* numpy: This library is used for numerical computing.

#### Code Overview

The code is designed to fetch geographic information associated with each node ID in a DataFrame of trip records. It utilizes the Overpass API to retrieve node and way data for each node ID.

#### Code Explanation

1. Initialization:
   * The required libraries (overpass, overpy, json, re, pandas, numpy) are imported.
   * An empty list dfs is created to store DataFrames for each trip.
2. Overpass API Initialization:
   * An instance of the overpy.Overpass class is created to interact with the Overpass API.
3. Data Retrieval Loop:
   * The code iterates over each row in the original DataFrame (df), representing trip records.
   * For each trip, it retrieves the trip ID and a list of node IDs associated with the trip.
4. Node Data Retrieval:
   * For each node ID, the code queries the Overpass API to retrieve geographic information.
   * It extracts latitude, longitude, tags, and associated way information for each node.
   * The retrieved information is stored in dictionaries and appended to a list (trip\_data).
5. Way Data Retrieval:
   * For each node, the code also queries ways associated with the node using the Overpass API.
   * It retrieves way IDs and associated tags for each way and stores them in the dictionaries.
6. DataFrame Construction:
   * For each trip, a DataFrame (trip\_df) is created from the list of dictionaries (trip\_data).
   * Each trip DataFrame is appended to the list dfs.
7. Concatenation:
   * Finally, all DataFrames in the dfs list are concatenated along rows to form a single DataFrame (new\_df).
8. Output:
   * The resulting DataFrame (new\_df) contains trip-wise node information, including trip ID, node ID, latitude, longitude, tags, way ID, and way tags.
   * The DataFrame is printed to the console for inspection.

#### Conclusion

This code efficiently retrieves geographic information for each node ID associated with trip records using the Overpass API. It constructs a comprehensive DataFrame suitable for further analysis and visualization of geographic data.

**Server.js \_\_\_\_\_\_\_\_\_\_\_**

code block defines several API endpoints using Express.js to handle requests related to trip data stored in a database. Below is an explanation of each endpoint:

1. GET /tripnote:
   * Purpose: Retrieves trip notes data from the trip table.
   * Query: Selects id, user\_id, purpose, start, stop, and n\_coord columns from the trip table.
   * Response: Returns the queried data as a JSON response.
2. GET /notescomments:
   * Purpose: Retrieves notes comments data from the noteinformation table.
   * Query: Selects user\_id, recorded, latitude, longitude, speed, details, text, and note\_type columns from the noteinformation table.
   * Response: Returns the queried data as a JSON response.
3. GET /usernote:
   * Purpose: Retrieves user notes data from the userinformation table.
   * Query: Selects user\_id, created, device, email, schoolZIP, workZIP, agetype, cycling\_freqtype, ethnicitytype, gendertype, incometype, rider\_historytype, and rider\_info columns from the userinformation table.
   * Response: Returns the queried data as a JSON response.
4. GET /trip:
   * Purpose: Retrieves trip data from the coord table for a specific trip\_id.
   * Query: Selects trip\_id, latitude, and longitude columns from the coord table based on a specified trip\_id.
   * Response: Returns the queried data as a JSON response.
5. GET /trip/:trip\_id:
   * Purpose: Retrieves trip data from the coord table for a specific trip\_id.
   * Parameters: Accepts a trip\_id as a route parameter.
   * Query: Selects trip\_id, latitude, and longitude columns from the coord table based on the provided trip\_id.
   * Response: Returns the queried data as a JSON response.
6. GET /trippagination:
   * Purpose: Retrieves paginated trip data from the osr table where the response starts with {"code": "Ok",.
   * Query: Selects trip\_id and response columns from the osr table, organizing the data by trip ID.
   * Parameters: Accepts page and limit query parameters for pagination.
   * Response: Returns the paginated trip data as a JSON response.
7. GET /trips:
   * Purpose: Retrieves trip data from the osr table with a default limit of 200 records.
   * Query: Selects trip\_id and response columns from the osr table, organizing the data by trip ID.
   * Response: Returns the queried trip data as a JSON response.

Each endpoint handles specific queries to fetch trip-related data from different tables in the database. These endpoints facilitate retrieving various types of trip information based on different criteria such as trip ID, pagination, and specific data attributes.

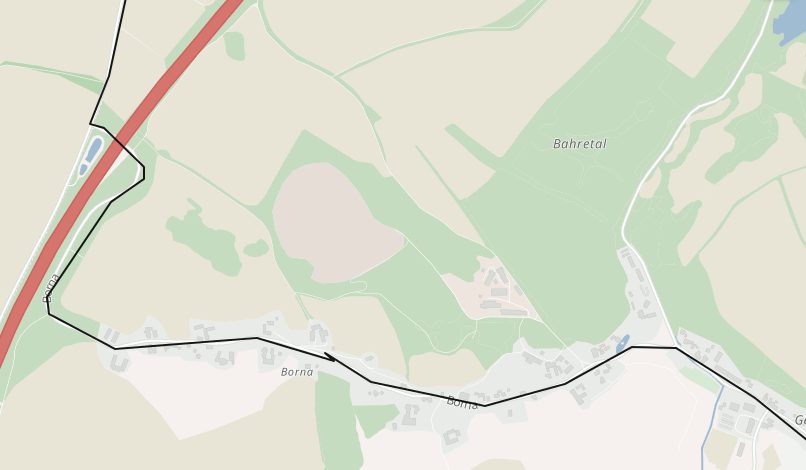
**Leaflet Routing Documentation - https://www.liedman.net/leaflet-routing-machine/api/**

Leaflet with OSRM - [https://www.liedman.net/leaflet-routing-machine/api/#l-routing-osrm](https://www.liedman.net/leaflet-routing-machine/api/" \l "l-routing-osrm)

<https://www.liedman.net/leaflet-routing-machine/tutorials/alternative-routers/> - For alternative router such as Graphhopper, Mapbox, Valhalla and other routing software

https://stackoverflow.com/questions/24919164/leaflet-polyline-smoothfactor-range

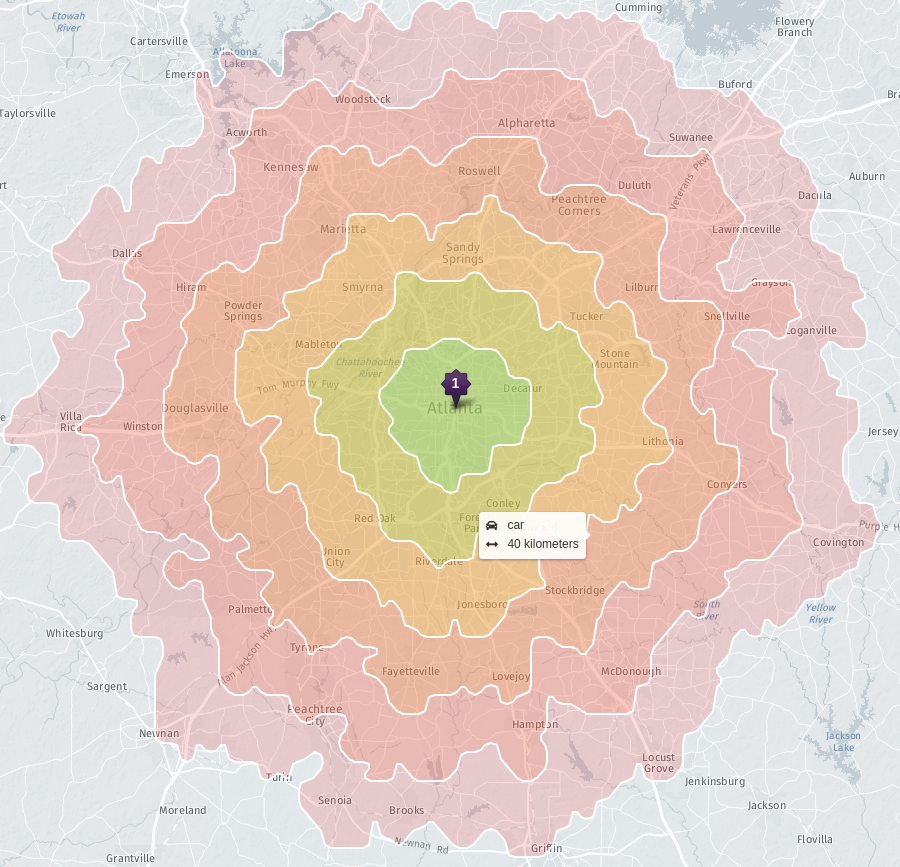
**Graphopper -** https://www.graphhopper.com/products/

[](https://docs.graphhopper.com/" \l "operation/postGPX)https://docs.graphhopper.com/#operation/postGPX

**Mapzen -** <https://www.mapzen.com/products/mobility/turn-by-turn/?d=0&lat=40.7259&lng=-73.9805&z=12&c=bicycle&st_lat=55.69&st_lng=12.595&st=Kastellet&end_lat=55.67&end_lng=12.593&end=Stadsgraven&use_bus=&use_rail=&use_transfers=&dt=&dt_type>=

**GIS OPS (valhalla) -** https://github.com/gis-ops

https://gis-ops.com/tutorials/

They are good for Isochrones - https://gis-ops.github.io/reachability-analysis/

**Alternative Routing Libraries**

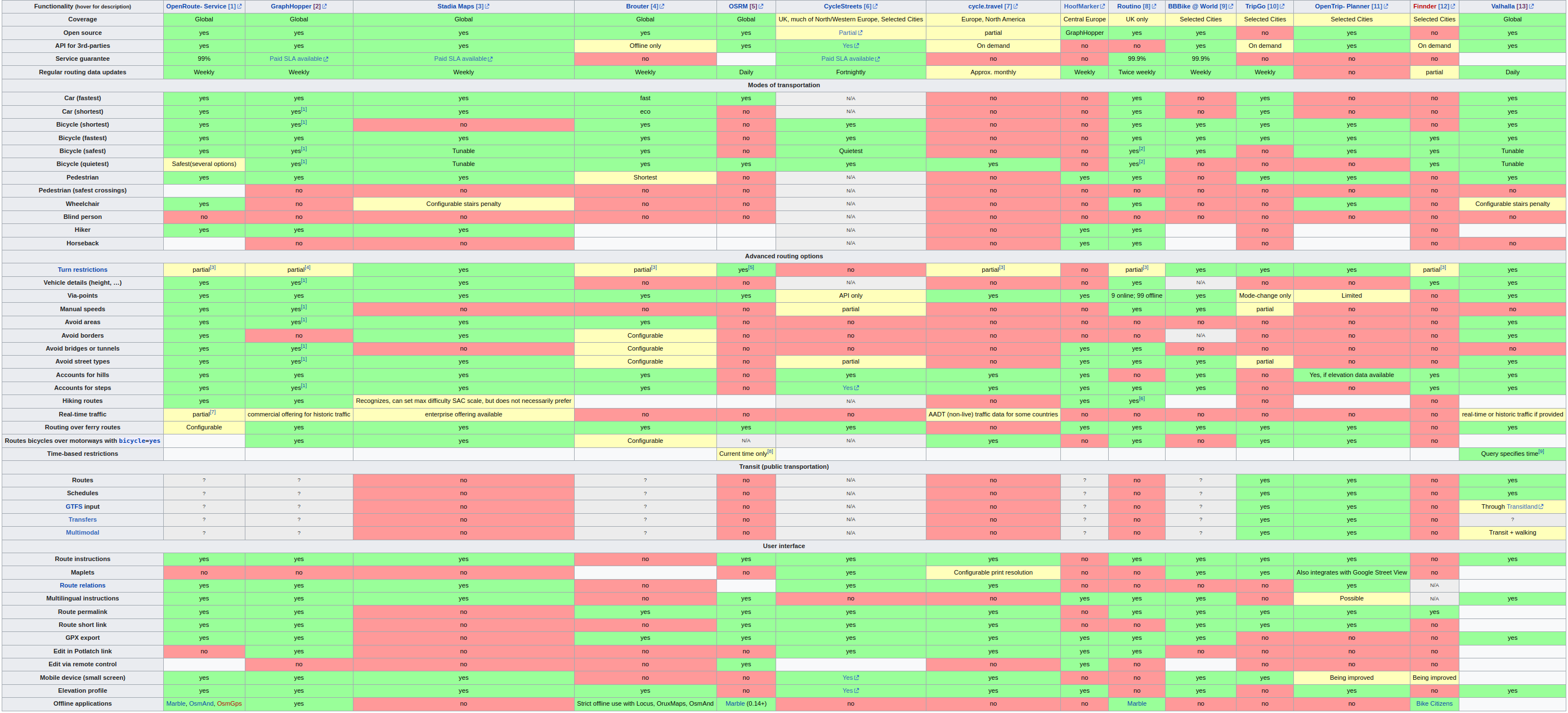
<https://github.com/makinacorpus/Leaflet.Snap> Leaflet Snap Plugin

https://turfjs.org/

Turf JS Tutorial - <https://www.youtube.com/watch?v=-i0M93W8yis>

**Alternative PostgreSQL Solutions**

https://postgis.net/



<https://wiki.openstreetmap.org/wiki/Routing/online_routers>

**Linux CMD Commands**

history | grep mysql – to check all the commands with mysql in it

history | grep docker - to check all the commands with docker in it

**To start MySQL:**

sudo mysql -u root -p (password is root)

**To start XAMPP:**

cd *opt*lampp/

sudo ./manager-linux-x64.run

**To access XAMPP/MariaDB through CMD:**

sudo /opt/lampp/bin/mysql -h localhost -u root -p

to input database: soruce …….filepath

**Download and Install XAMPP -** <https://www.youtube.com/watch?v=HJl2ILUfBoA>

<https://linux.how2shout.com/how-to-start-xampp-in-ubuntu-using-the-command-line/>

**Download Docker CE and Docker Desktop -** <https://www.youtube.com/watch?v=ILdziITdSag>

<https://www.youtube.com/watch?v=JsXNBIsFzu4>

**Few Common Erros Faced:**

Require Error:

*const express = require("express"); ^ ReferenceError: require is not defined in ES module scope, you can use import instead This file is being treated as an ES module because it has a '.js' file extension and '/backend/package.json' contains "type": "module". To treat it as a CommonJS script, rename it to use the '.cjs' file extension.*

<https://github.com/vercel/next.js/issues/24334>

<https://learn.coderslang.com/0021-nodejs-require-is-not-defined-error/>

**Open GraphHopper on Localhost:**

docker run -p 8989:8989 -v "$(pwd)/data:/data" israelhikingmap/graphhopper --input /data/georgia-latest.osm.pbf --host 0.0.0.0 --config /data/config-graphhopper.yml

**Youtube Tutorials for References**

<https://www.youtube.com/watch?v=F8dnYNTncoU&t=1757s> –LLeaflet Map Routing JavaScript App

<https://www.youtube.com/watch?v=PMtXhxW6t2k> - Leaflet - Build React.js Map App’

<https://www.youtube.com/watch?v=nZaZ2dB6pow> Mapping Geolocation with Leaflet.js - Working with Data and APIs in JavaScript

<https://www.youtube.com/watch?v=H91aqUHn8sE> – Node and TypeScript